INPI
INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

BREVET D'INVENTION

CERTIFICAT D'UTILITÉ - CERTIFICAT D'ADDITION

COPIE OFFICIELLE

Le Directeur général de l'Institut national de la propriété industrielle certifie que le document ci-annexé est la copie certifiée conforme d'une demande de titre de propriété industrielle déposée à l'Institut.

Fait à Paris, le 22 SEP. 2004

Pour le Directeur général de l'Institut
national de la propriété industrielle
Le Chef du Département des brevets

Martine PLANCHE

UMENT DE PRIORITÉ

SENTÉ OU TRANSMIS
NFORMÉMENT À LA
RÈGLE 17.1.a) OU b)

INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE
SIEGE
26 bis, rue de Saint-Petersbourg
75800 PARIS cedex 08
Téléphone : 33 (0)1 53 04 53 04
Télécopie : 33 (0)1 53 04 45 23
www.inpi.fr

INPI
INSTITUT NATIONAL DE LA PROPRIÉTÉ INDUSTRIELLE

26 bis, rue de Saint Pétersbourg - 75800 Paris Cedex 08

Pour vous informer : INPI DIRECT
N° Indigo 0 825 83 85 87
0,15 € TTC/mn

Télécopie : 33 (0)1 53 04 52 65

BREVET D'INVENTION
CERTIFICAT D'UTILITÉ

Code de la propriété intellectuelle - Livre VI

N° 11354*03

REQUÊTE EN DÉLIVRANCE
page 1/2

BR1

Cet imprimé est à remplir lisiblement à l'encre noire

DB 540 @ W / 030103

Réservé à l'INPI

REMISE DES PIÈCES
DATE 4 SEPT 2003
LIEU 75 INPI PARIS

N° D'ENREGISTREMENT NATIONAL ATTRIBUÉ PAR L'INPI 0310474

DATE DE DÉPÔT ATTRIBUÉE PAR L'INPI 22 SEP. 2003

Vos références pour ce dossier *(facultatif)*

1 NOM ET ADRESSE DU DEMANDEUR OU DU MANDATAIRE À QUI LA CORRESPONDANCE DOIT ÊTRE ADRESSÉE

Joseph Michel Koskas
63, rue de la Colonie
75013 Paris

Confirmation d'un dépôt par télécopie ☐ N° attribué par l'INPI à la télécopie

2 NATURE DE LA DEMANDE — **Cochez l'une des 4 cases suivantes**

Demande de brevet	☒	
Demande de certificat d'utilité	☐	
Demande divisionnaire	☐	
Demande de brevet initiale	N°	Date
ou demande de certificat d'utilité initiale	N°	Date
Transformation d'une demande de brevet européen *Demande de brevet initiale*	☐ N°	Date

3 TITRE DE L'INVENTION (200 caractères ou espaces maximum)

A Database Management Algorithm

4 DÉCLARATION DE PRIORITÉ OU REQUÊTE DU BÉNÉFICE DE LA DATE DE DÉPÔT D'UNE DEMANDE ANTÉRIEURE FRANÇAISE

Pays ou organisation
Date N°
Pays ou organisation
Date N°
Pays ou organisation
Date N°

☐ **S'il y a d'autres priorités, cochez la case et utilisez l'imprimé «Suite»**

5 DEMANDEUR (Cochez l'une des 2 cases) — ☐ **Personne morale** ☒ **Personne physique**

Nom ou dénomination sociale		Koskas
Prénoms		Joseph, Michel
Forme juridique		
N° SIREN		
Code APE-NAF		
Domicile ou siège	Rue	63, rue de la Colonie
	Code postal et ville	75013 Paris
	Pays	France
Nationalité		franaise
N° de téléphone *(facultatif)*		N° de télécopie *(facultatif)*
Adresse électronique *(facultatif)*		

☐ **S'il y a plus d'un demandeur, cochez la case et utilisez l'imprimé «Suite»**

Remplir impérativement la 2ème page

INPI
INSTITUT NATIONAL DE LA PROPRIÉTÉ INDUSTRIELLE

BREVET D'INVENTION
CERTIFICAT D'UTILITÉ

REQUÊTE EN DÉLIVRANCE
page 2/2

BR2

Réservé à l'INPI

REMISE DES PIÈCES
DATE 4 SEPT 2003
LIEU 75 INPI PARIS
N° D'ENREGISTREMENT NATIONAL ATTRIBUÉ PAR L'INPI 0310474

DB 540 W / 210502

6 MANDATAIRE *(s'il y a lieu)*		
Nom		
Prénom		
Cabinet ou Société		
N °de pouvoir permanent et/ou de lien contractuel		
Adresse	Rue	
	Code postal et ville	
	Pays	
N° de téléphone *(facultatif)*		
N° de télécopie *(facultatif)*		
Adresse électronique *(facultatif)*		
7 INVENTEUR (S)		**Les inventeurs sont nécessairement des personnes physiques**
Les demandeurs et les inventeurs sont les mêmes personnes		☒ Oui ☐ Non : **Dans ce cas remplir le formulaire de Désignation d'inventeur(s)**
8 RAPPORT DE RECHERCHE		**Uniquement pour une demande de brevet (y compris division et transformation)**
Établissement immédiat ou établissement différé		☐ ☒
Paiement échelonné de la redevance *(en deux versements)*		**Uniquement pour les personnes physiques effectuant elles-mêmes leur propre dépôt** ☒ Oui ☐ Non
9 RÉDUCTION DU TAUX DES REDEVANCES		**Uniquement pour les personnes physiques** ☐ Requise pour la première fois pour cette invention *(joindre un avis de non-imposition)* ☐ Obtenue antérieurement à ce dépôt pour cette invention *(joindre une copie de la décision d'admission à l'assistance gratuite ou indiquer sa référence)* : AG
10 SÉQUENCES DE NUCLEOTIDES ET/OU D'ACIDES AMINÉS		☐ Cochez la case si la description contient une liste de séquences
Le support électronique de données est joint		☐
La déclaration de conformité de la liste de séquences sur support papier avec le support électronique de données est jointe		☐
Si vous avez utilisé l'imprimé «Suite», indiquez le nombre de pages jointes		

11 SIGNATURE DU DEMANDEUR OU DU MANDATAIRE
(Nom et qualité du signataire)
Koskas, inventeur

VISA DE LA PRÉFECTURE OU DE L'INPI
M. ROCHET

La loi n°78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés s'applique aux réponses faites à ce formulaire.
Elle garantit un droit d'accès et de rectification pour les données vous concernant auprès de l'INPI.

Un algorithme hiérarchique de gestion de bases de données

Michel Koskas*

Résumé

L'invention présentée ici est un algorithme permettant une gestion complète de grandes bases de données. Il repose sur une représentation dénormalisée de la base ainsi que sur le stockage d'ensemble d'entiers dans des arbres à radicaux.

1 Introduction

Cette invention est une nouvelle façon de gérer des bases de données. Il repose sur l'utilisation d'arbres à radicaux pour stocker des ensembles d'entiers et pour faire des opérations logiques dessus.

La complexité de l'algorithme présenté ici est plus basse que celle des algorithmes couramment utilisés. Ces derniers consomment en effet de grandes ressources machine, que ce soit du point de vue des ressources processeurs que des ressources disques. La présente invention pallie ces inconvénients.

L'algorithme utilise en outre une représentation hiérarchique des données.

2 Arbres à radicaux

Un arbre à radicaux est un moyen pratique de stocker des ensembles d'entiers, particulièrement quand ils sont écrits sur une même longueur. Lorsqu'on utilise des entiers, il est clairement toujours possible de leur imposer la même longueur d'écriture (celle du plus long ou plus) en faisant précéder leur écriture du nombre adéquat de chiffres 0.

Considérons par exemple un ensemble d'entiers qu'on écrit sur une longueur unique en base 2, $S = \{0, 2, 5, 7, 11\} = \{0000, 0010, 0101, 0111, 1011\}$.

*63, rue de la Colonie 75013 Paris joseph.koskas@9online.fr

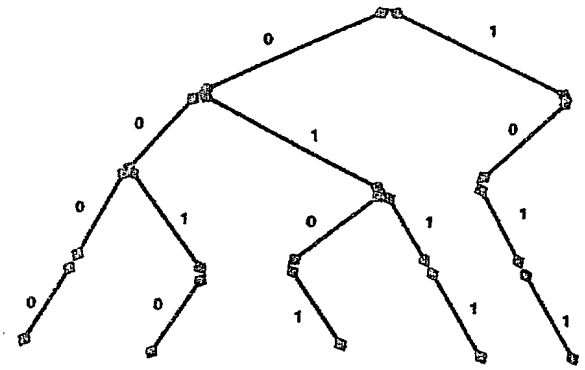

FIG. 1 – L'ensemble $\{0, 2, 5, 7, 11\}$ stocké dans un arbre à radicaux

On peut alors stocker cet ensemble dans un arbre à radicaux dont les chemins, depuis la racine jusqu'aux feuilles, représente l'écriture de l'entier stocké dans la feuille de l'arbre. Par exemple, l'ensemble précédent peut être stocké dans l'arbre à radicaux suivant :

Les avantages à utiliser un arbre à radicaux sont fort nombreux : le stockage est économique en termes de place mémoire car les préfixes communs aux nombres distincts ne sont stockés qu'une seule fois. Par ailleurs, comme nous le verrons dans les paragraphes suivants, les opérations logiques sur des ensembles stockés ainsi sont rapides, économiques en termes de ressources machines et simples à implémenter.

2.1 Intersection

Si on considère deux ensembles d'entiers, stockés dans des arbres à radicaux, le coût du calcul de l'intersection de ces deux ensembles est $O(ih)$ où i est le cardinal de l'intersection de ces deux ensembles et h la plus petite des deux hauteurs des deux arbres qu'on intersecte. Il suffit en effet de parcourir les deux arbres en largeur simultanément.

2.2 Union

De même le calcul de la réunion de deux ensembles stockés dans des arbres à radicaux est $O(uh)$ où u est le cardinal de la réunion et h la plus grande des deux hauteurs des deux arbres dont on calcule l'union.

3 Création des indexes

Dans cette section, on détaille comment des arbres à radicaux peuvent être utiles et efficaces pour stocker les données d'une base de données ou

pour la modifier.

Dans la première sous-section, on suppose que la base de données n'est composée que d'une table, elle-même composée d'une unique colonne.

Puis nous supposerons que la base de données n'est composée que d'une unique table, elle-même composée de plusieurs colonnes, et d'au moins une clé primaire. Il peut en effet s'avérer très pratique d'autoriser une table à être munie de plusieurs clés primaires. En effet, en pratique, il arrive fréquemment qu'une ligne de table ne soit que partiellement remplie. Il se peut alors qu'une des clés primaires soit incomplète, donc inutilisable, mais qu'une autre soit complète.

La dernière sous-section est dédie à la création des indexes d'une bases de données quelconque.

3.1 une table, une colonne

Clés primaires. Une clé primaire est une colonne, ou un ensemble ordonné de colonnes, tel que deux lignes différentes de la table ne puisse prendre les mêmes valeurs sur cette (ou ces) colonnes.

Il existe cependant toujours une clé primaire implicite et très utile : l'indice de la ligne dans la table (il s'agit en effet d'une clé primaire puisque deux lignes distinctes ne peuvent avoir le même indice de ligne). Dans la suite de la description de cette invention, nous supposerons que cette clé primaire est effective.

Si on doit stocker, interroger et gérer une base de données faite d'une unique table elle-même constituée d'une unique table, on peut calculer le thesaurus de cette colonne et pour chaque mot de ce thesaurus calculer l'ensemble des indices de lignes auxquels il apparaît.

Ces indices de lignes peuvent tout naturellement être stockés dans un arbre à radicaux.

3.1.1 Création du thesaurus

Remarquons qu'au cours de la création du thesaurus, un tri des données est effectué. On trie en effet l'ensemble des couples (mot, indice de ligne) selon les mots et, à mot égal, selon les indices de lignes. Ainsi on peut d'une part calculer le thesaurus et d'autre part, pour chacun des mots de ce thesaurus l'arbre à radicaux des indices de lignes auxquels il apparaît.

Prenons un exemple : la table :

0	Male
1	Female
2	Female
3	Male
4	Female
5	Male
6	Male
7	Female
8	Female
9	Male
10	Male

(dans cet exemple, les indices de lignes sont indiqués explicitement).

On construit alors les couples (Male, 0), (Female, 1), (Female, 2), (Male, 3), (Female, 4), (Male, 5), (Male, 6), (Female, 7), (Female, 8), (Male, 9), (Male, 10) et on les trie selon leur premier élément en priorité :

(Female, 1), (Female, 2), (Female, 4), (Female, 7), (Female, 8), (Male, 0), (Male, 3), (Male, 5), (Male, 6), (Male, 9), (Male, 10).

On peut alors construire le thesaurus et, pour chaque mot du thesaurus, l'ensemble des indices de lignes auxquels il apparaît.

Le mot "Female" apparaît aux lignes $\{1, 2, 4, 7, 8\}$ et "Male" apparaît aux indices $\{0, 3, 5, 6, 9, 10\}$.

Après ce travail, il est très simple de répondre à une question comme "Quels sont les indices de lignes auxquels apparaît le mot "Male" ?" mais relativement difficile de répondre à une question comme "Quel est le contenu de la cellule de la ligne 5 ?" . Pour ce genre de requête, on pourra consulter la sous-section 5 ci-après.

3.1.2 Stockage des fonctions indicatrices

Ces ensembles d'indices de lignes peuvent donc être stockés dans des arbres à radicaux. Ce procédé de stockage est très utile pour calculer l'intersection, la réunion etc. . . .de tels ensembles.

Dans l'exemple précédent, on obtient (voir figure 2) :

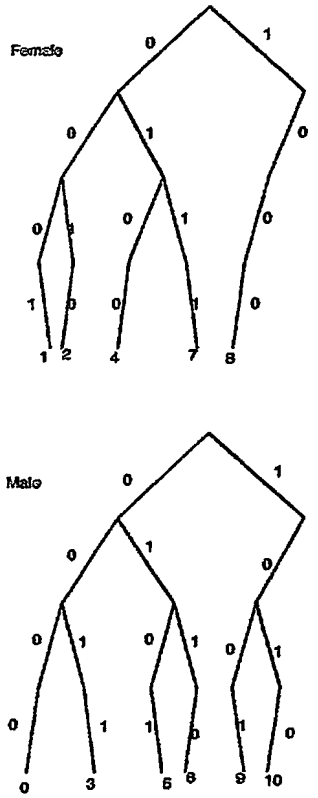

Figure 2: Example of the representation of a column of a table

3.1.3 Macro mots

Il est une autre requête courante qui concerne le contenu d'une colonne : le "entre" : on peut souhaiter connaître les indices de lignes dont le contenu est compris entre deux bornes données.

Imaginons par exemple qu'une colonne contienne des dates, écrites au format AAAAMMJJ. Comparer deux dates stockées sous ce format est en fait la même chose que les comparer lexicographiquement.

Mais nous pouvons aussi enrichir le thesaurus de mots obtenus comme troncatures des mots du thesaurus initial. Par exemple, on peut décider d'enrichir le thesaurus de toutes les troncatures des quatre ou six premières lettres des mots du thesaurus initial.

Ainsi chaque mot serait représenté, dans notre exemple, trois fois : une fois en tant que lui même, une fois tronqué à six caractères et une dernière fois tronqué à quatre caractères.

Tout mot de six caractères, disons aaaamm, apparaîtra à chaque fois que la ligne initiale contenait un mot aaaammxx. En d'autres termes, l'ensemble des lignes auxquelles apparaîtra le mot aaaamm est la réunion des ensembles d'indices de lignes où apparait un mot de la forme aaaammxx (c'est-à-dire aaaamm suivi de quoi que ce soit).

De même un mot de quatre caractères aaaa apparaîtra à chaque fois qu'un mot de la forme aaaaxxyy était présent dans la table initiale. Son arbre à radicaux est donc l'union des arbres à radicaux des mots dont il est préfixe.

L'intérêt est qu'une clause "entre" (between en anglais) peut se traiter avec une économie importante en termes de lectures sur procédé de stockage. Par exemple, si on cherche l'ensemble des lignes auxquelles apparaît une date comprise dans l'intervalle [19931117, 19950225], le nombre de lectures nécessaires d'arbres à radicaux est de $14 + 1 + 1 + 1 + 25 = 42$ (car $[19931117, 19950225] = [19931117, 19931130] \cup [199312, 199312] \cup [1994, 1994] \cup [199501, 199501] \cup [10050201, 19950225]$), au lieu de 466.

3.1.4 Gérer les manques

Il peut parfois se trouver que certaines lignes d'une tables ne soient pas renseignées. Mais pour créer les arbres à radicaux, toute ligne devrait avoir une valeur. On choisit donc des valeurs signifiant que la ligne correspondante ne contient pas d'information. Naturellement, on choisira une valeur ayant un rapport avec le type des données stockées; À titre d'exemple, on peut choisir : $\#Empty\#$ pour une chaîne de caractères, -2^{31} pour un entier signé sur 32 bits, $2^{32} - 1$ pour un entier non signé sur 32 bits, -2^63 pour un entier signé sur 64 bits, $2^{64} - 1$ pour un entier non signé sur 64 bits, etc....

3.1.5 Un stockage supplémentaire

Comme expliqué précédemment, le stockage d'une colonne par thesaurus et arbres à radicaux n'est pas très pratique pour répondre à une requête comme "Quelle est la valeur de la ligne 17 ?". par exemple.

C'est pourquoi il est nécessaire de stocker en sus la colonne dans son ordre naturel. Bien sûr, plutôt que stocker la colonne elle-même, il sera souvent avantageux de stoker la suite des indices de mots dans le thesaurus.

Par exemple, la colonne précédente sera stockée de la façon suivante :

0	Male
1	Female
2	Female
3	Male
4	Female
5	Male
6	Male
7	Female
8	Female
9	Male
10	Male

sera stockée en :

Thesaurus	
0	Female
1	Male

et la colonne :

0
1
1
0
1
0
0
1
1
0
0

Remarque : il peut se trouver qu'au fur et à mesure que la base de données est transformée, un mot apparaisse ou disparaisse du thesaurus (par exemple lorsqu'on retire ou ajoute des lignes à la table). On pourrait dès lors penser que la réécriture complète de la colonne est nécessaire. Ce n'est en fait pas le cas : Plutôt que stocker le thesaurus trié, on peut le stocker non trié et enregistrer à part une permutation permettant de retrouver l'ordre lexicographique des mots qui le composent. C'est pourquoi si un mot apparaît dans le thesaurus, la réécriture complète de la colonne n'est pas nécessaire. On réécrit dans ce cas la permutation permettant de retrouver l'ordre lexicographique des mots plutôt que le thesaurus lui-même.

3.1.6 Résumé du stockage complet d 'une colonne

3.2 Une table, plusieurs colonnes

Dans le cas d'une base de données ne contenant qu'une seule table elle-même constituée de plusieurs colonne peut être traitée comme si elle était formée de colonnes indépendantes. En d'autres termes, on peut créer le stockage de chacune des colonnes constituant la table.

La seule question en suspens est alors le traitement de la clé primaire.

Lorsqu'on s'occupe d'une clé primaire, on a besoin d'être capable de répondre le plus rapidement possible à deux types de requêtes opposées : "À quelle ligne peut-on trouver une valeur données de clé primaire" et "Quelle est la valeur de la clé primaire trouvée à une ligne donnée ?".

On peut répondre efficacement à ces deux types d'interrogation en stockant à la fois la colonne ou les colonnes formant la clé primaire dans l'ordre

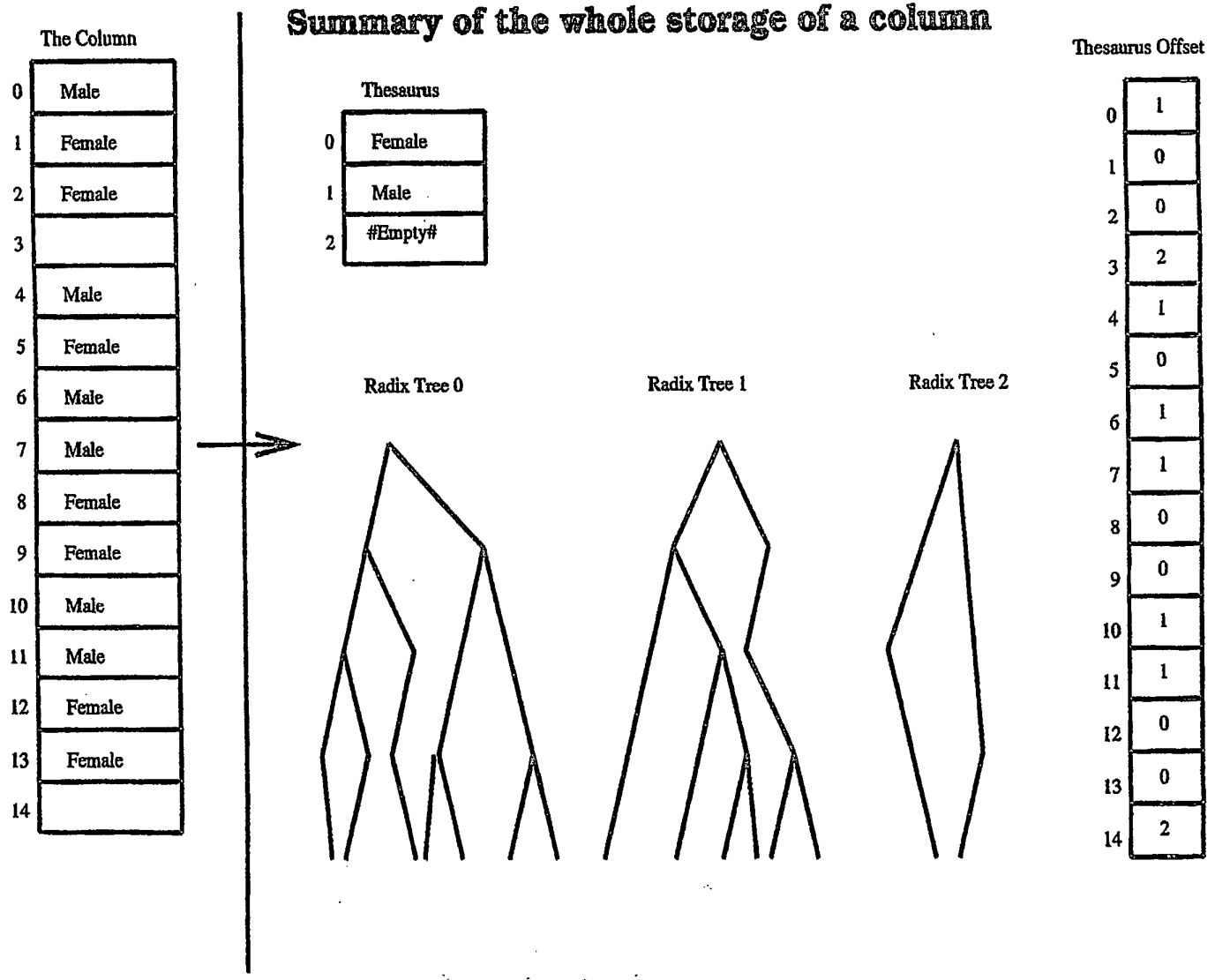

Figure 3: Résumé du stockage complet d'une colonne

dans lequel les lignes apparaissent dans la table et une permutation permettant de lire les colonnes dans l'ordre lié à une fonction quelconque de comparaison. On peut alors retrouver une valeur donnée par dichotomie.

Par exemple, imaginons qu'une clé primaire est formée de deux colonnes, dont les valeurs sont stockées dans le tableau ci-dessous.

(0)	1	3
(1)	2	1
(2)	3	2
(3)	2	3
(4)	1	2
(5)	3	7
(6)	2	2
(7)	1	1
(8)	3	3
(9)	4	3

Dans cet exemple, les indices de lignes sont à nouveau exprimés explicitement mais mis entre parenthèses. On stocke donc les deux colonnes exactement comme elles sont dans la table et une permutation, liée à une fonction de comparaison qu'on choisit. Par exemple, on peut décider de comparer d'abord la première colonne lexicographiquement et à valeur égale comparer la deuxième ordinalement.

Dans ce cas, la clé primaire triée est :

(7)	1	1
(4)	1	2
(0)	1	3
(1)	2	1
(6)	2	2
(3)	2	3
(2)	3	2
(8)	3	3
(5)	3	7
(9)	4	3

En retirant les valeurs (mais en gardant les indices), on obtient la permutation (7401632859). La plus petite valeur se trouve donc à la ligne 7, la deuxième plus petite à la ligne 4 etc. ...Retrouver une valeur donnée se fait ainsi facilement par dichotomie.

Lorsqu'on stocke une table, il est très pratique de stocker et maintenir à jour le nombre total de lignes dont elle est constituée.

3.3 Plusieurs tables

Dans une base de données relationnelle, il y a habituellement plusieurs tables reliées entre elles par des jeux de clés primaires, clés étrangères.

Comme expliqué précédemment, une clé primaire est une colonne ou un ensemble ordonné de colonnes ne pouvant prendre la ou les mêmes valeurs en deux lignes distinctes. (L'indice de ligne est un exemple fondamental de clé primaire.)

Supposons qu'une table soit constituée de plusieurs millions de lignes, mais que certaines de ses colonnes ne puissent prendre qu'un nombre très réduit de valeurs différentes (par exemple, une base de données contenant des données de généalogie peut contenir les noms de personnes, pour chacune d'elles son pays de naissance, son continent de naissance, les pays et continents de naissance de sa mère et de son premier enfant s'il en a. Au lieu de renseigner toutes ces colonnes, il est considéré comme très économique de stocker dans un tel cas les pays dans une table séparée de la table principale et le continents dans une troisième table. La table principale contient alors à chaque ligne une valeur (une clé étrangère) donnant un identifiant de ligne (une valeur de clé primaire) de la table "pays" et la table "pays" contient, à chacune de ses lignes, une valeur (une clé étrangère) identifiant une des lignes de la table "continent" (clé primaire).

Voici un exemple miniature illustrant ce qui précède :

(li)	cn	Inc	BirCoun	BirCont	MoCoun	MoCont	EldCoun	EldCont
(0)	Dupont	817	France	Europe	Tunisia	Africa	England	Europe
(1)	Gracamoto	1080	Japan	Asia	Japan	Asia	USA	America
(2)	Smith	934	England	Europe	India	Asia	England	Europe
(3)	Helmut	980	Germany	Europe	Germany	Europe	Germany	Europe

(dans cet exemple, "cn" désigne le nom, "inc" le revenu, "BirCoun" le pays de naissance, "BirCont" le continent de naissance, "MoCoun" le pays de naissance de la mère, "MoCont" le continent de naissance de la

mère, "EldCoun" le pays de naissance de l'aîné et "EldCont" le continent de naissance d'aîné.

Cette table peut être réécrite en plusieurs tables :

Continents :

Continent	
(li)	Continent
(0)	Africa
(1)	America
(2)	Asia
(3)	Europe

Pays :

Country		
(li)	Country	Continent
(0)	France	3
(1)	Tunisia	0
(2)	England	3
(3)	Japan	2
(4)	USA	1
(5)	India	2
(6)	Germany	3

La table principale devient ainsi :

Customers					
(li)	cn	Inc	BirCoun	MoCoun	EldCoun
(0)	Boyer	817	0	1	2
(1)	Gracamoto	1080	3	3	4
(2)	Smith	934	2	5	2
(3)	Helmut	980	6	6	6

L'ensemble des trois tables ainsi construites occupe certes moins de place que la table initiale.

Mais cela illustre aussi l'idée selon laquelle une base relationnelle peut être transformée en un ensemble de tables indépendantes les unes des autres.

Dans l'exemple précédent, on peut considérer la table "Continents" par elle-même, la table "Pays" avec la table "continent" développée dedans (c'est-à-dire la table "pays" dans laquelle les références à la table "continents" ont été remplacées par les lignes de la table elle-même) et la table "Personnes" avec les tables "Pays" et "Continents" développées dedans.

Les tables d'expansion sont alors :

développed Continents	
(li)	Continent
(0)	Africa
(1)	America
(2)	Asia
(3)	Europe

développed		**Countries**
(li)	Country	Continent
(0)	France	Europe
(1)	Tunisia	Africa
(2)	England	Europe
(3)	Japan	Asia
(4)	USA	America
(5)	India	Asia
(6)	Germany	Europe

développed Customers								
(li)	cn	Inc	BirCoun	BirCont	MoCoun	MoCont	EldCoun	EldCont
(0)	Boyer	817	France	Europe	Tunisia	Africa	England	Europe
(1)	Gracamoto	1080	Japan	Asia	Japan	Asia	USA	America
(2)	Smith	934	England	Europe	India	Asia	England	Europe
(3)	Helmut	980	Germany	Europe	Germany	Europe	Germany	Europe

Il peut bien évidemment arriver, comme dans cet exemple, qu'une table soit amenée à se développer plusieurs fois dans une autre. Cela signifie qu'une colonne de table développée dans une autre devra toujours être référencée comme appartenant à la table d'expansion, développée dans la table d'expansion via un jeu de clés primaires et clés étrangères qu'il feront partie de l'identité de la colonne.

Nous définissons donc une table d'expansion comme une table dans laquelle toutes les tables qui pouvaient être développées en elle l'ont été, en autant d'exemplaires qu'il y a de jeux de clés primaires clés étrangères permettant de passer de la table d'expansion à la table développée. Désormais, nous considérerons donc que la base de données relationnelle est formée de tables d'expansion indépendantes les unes des autres.

Pour chacune de ces tables d'expansion, on peut bâtir les indexes comme expliqué dans le cas d'une table seule.

Nous sommes maintenant en mesure d'interroger et de modifier notre base de données ainsi indexée.

4 Le requêtage

Dans cette section nous expliquons comment les indexes créés peuvent être utilisés pour résoudre efficacement des requêtes SQL. Habituellement, une requête implique plusieurs tables et peut être séparée en deux parties distinctes : la clause "where" qui demande au gestionnaire de bases de données de calculer des indices de lignes d'une table et une partie demandant au gestionnaire de bases de données de faire des calculs sur des données se trouvant aux lignes calculées.

La première partie peut contenir des jointures de tables (un lien entre une clé étrangère et la clé primaire correspondante), une comparaison entre une colonne et une constante (avec un connecteur arithmétique comme $=$, $\geq$, $>$, $<$, $\leq$, between , like, in...) ou une comparaison entre deux colonnes (mêmes opérateurs arithmétiques ou encore un produit cartésien). Ces requêtes sont connectées entre elles, lorsqu'il y en a plusieurs, par des connecteurs logiques (and, or, not...).

La deuxième partie de la requête peut contenir des opérations arithmétiques comme des sommes, des moyennes, des produits, l'opérateur de dénombrement * etc....

4.1 Traitement des clauses de jointures : choix de la table d'expansion

Comme expliqué précédemment, chacune des tables est considérée comme table d'expansion, ce qui signifie que les jointures de tables ne sont pas pertinentes pour une telle table.

Mais une requête comporte habituellement plusieurs tables. Comment choisir la table d'expansion dans laquelle résoudre la requête ?

Les tables impliquées dans la requête sont toutes développées dans un ensemble non vide, disons T. Une seule de ces table d'expansion n'est pas développée dans les autres. Cette table est la table dans laquelle nous devons résoudre la requête.

La clause "where" contient donc des clauses de jointures, reliées logiquement au reste de la requête par des connections logiques "et". Il suffit donc de tout simplement les effacer en remplaçant toute la clause "et" par l'autre terme. cela signifie qu'on remplace "(Jointure ET Reste)" par "Reste" et ce pour chaque clause de jointure.

Voyons à présent comment gérer efficacement une clause "where" débarrassée de ses clauses de jointure.

4.2 Requêtes atomiques

Nous appelons "requête atomique" une portion indivisible de la clause where, c'est-à-dire une comparaison qui forme toute la requête ou qui est reliée au reste de la requête par des connecteurs logiques sans qu'elle en comporte elle-même. Si une table t comporte la colonne c, une requête atomique sera par exemple `t.c = 3`, `t.c between ''HIGH'' and ''MEDIUM''`, ou encore `t.c like Word%`.

Les prochains sous-paragraphes expliquent comment gérer les requêtes atomiques.

4.2.1 Cas d'égalité entre une colonne et une valeur donnée

Ce cas est le plus simple à traiter. Il suffit de lire l'arbre à radicaux de la valeur recherchée pour la colonne de la requête.

4.2.2 clause Between

Ceci est un exemple fondamental de requête atomique. Toutes les autres requêtes atomiques peuvent se ramener à ce cas. C'est pour cette clause que les macro-mots ont été créés.

Reprenons l'exemple des dates donné dans le paragraphe sur les macro mots. Cette colonne a été gérée en enrichissant le vocabulaire des troncatures de ses mots de longueur 4 et de longueur 6. Si on cherche les lignes dont la valeur est comprise entre [19931117, 19950225], il suffit de découper l'intervalle en : [19931117, 19950225] = [19931117, 19931130] ∪ [199312, 199312] ∪ [1994, 1994] ∪ [199501, 199501] ∪ [10050201, 19950225].

Le calcul est alors très simple : on lit l'arbre à radicaux de la valeur 19931117, qu'on unit (opération logique "or") avec celui de la valeur 19931118, ...qu'on unit avec celui de la valeur 19931130 puis avec celui de la valeur (tronquée à 6 caractères) de 199312 puis avec celui (tronqué à 4 caractères) de 1994 puis avec celui (tronqué à 6 caractères) de 199501, puis avec celui de 19950201 puis avec celui de ...19950225.

Ainsi est-on amené à lire 42 arbres à radicaux au lieu des 466 qu'il aurait fallu lire sans les macro mots.

Le traitement des "or" est expliqué plus bas.

On peut bien sûr traiter des intervalles semi ouverts ou ouverts en excluant simplement les mots correspondants.

4.2.3 Supérieur ou égal, inférieur ou égal, supérieur, inférieur

Chacune de ces requêtes atomiques est une clause between qui s'ignore. En effet, si on appelle m et M le minimum et maximum du thesaurus de la colonne concernée, alors

`t.c > a`	signifie	`t.c between ]a, M]`
$t.c \geq a$	signifie	`t.c between [a, M]`
`t.c < a`	signifie	`t.c between [m, a[`
$t.c \leq a$	signifie	`t.c between [m, a]`

On peut ainsi traiter ces clauses comme une clause between.

4.2.4 Clause In

La clause In est une façon de mixer des égalités par des "or". On peut donc les gérer très simplement.

Par exemple, `t.c in (a,b,c)` peut se réécrire en `t.c = a or t.c = b or t.c = c`. La gestion des clauses "or" est expliquée plus bas.

4.2.5 Like

La clause "like" est un autre exemple de clause between. Par exemple, la clause `t.c like Mot%` se réécrit en effet en `t.c between [Mot, Mou[`.

4.3 Connections entre requêtes atomiques

Les requêtes atomiques peuvent être mixées à l'aide de connecteurs logiques : le "And", le "Or" et le "Not". Les trois prochaines sous-sections sont dédiées à ces opérateurs.

Nous souhaitons d'abord insister sur le fait qu'une requête atomique renvoie toujours en arbre à radicaux, ce qui sera aussi le cas de ces opérateurs logiques et pour finir de la clause where.

4.3.1 Or

Le fait de faire un "or" ("ou") de deux arbres à radicaux est en fait l'opération consistant à les réunir. Ce calcul peut très aisément se faire par un parcours en largeur des deux arbres simultanément. Il se fait récursivement par

```
union(t1, t2)
Debut
Arbre res;
Si (t1 = NULL) res = t2
Si (t2 = NULL) res = t1
res->FilsGauche = Union(t1->FilsGauche, t2->FilsGauche)
res->FilsDroit = Union(t1->FilsDroit, t2->FilsDroit)
Renvoyer res
Fin
```

4.3.2 And

Cette clause se calcule presque comme la précédente (elle correspond à une intersection) :

```
Intersection(t1, t2)
Debut
Arbre res;
Si (t1 = NULL) res = NULL
Si (t2 = NULL) res = NULL
res->FilsGauche = Intersection(t1->FilsGauche, t2->FilsGauche)
res->FilsDroit = Intersection(t1->FilsDroit, t2->FilsDroit)
renvoyer res
Fin
```

Cette clause demande néanmoins moins de temps de calcul en moyenne que la précédente. En effet lors des deux parcours des arbres en parallèle il suffit qu'un des deux noeuds explorés n'aie pas de fils gauche pour que l'exploration du fils gauche de l'autre noeud soit inutile.

Cela est particulièrement vrai quand les arbres ont été stockés sur disque dur dans des fichiers séparés.

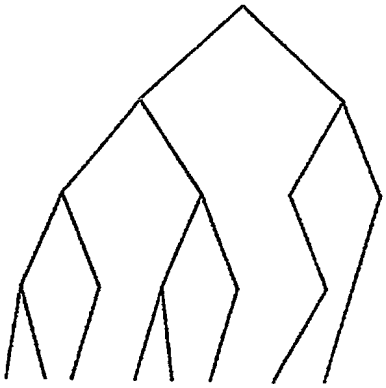

FIG. 4 – A 13-arbre à radicaux avant le NOT

4.3.3 Not

Cette clause est une des plus difficiles à mettre en oeuvre parmi les requêtes atomiques. Elle peut néanmoins se traiter assez facilement.

L'indice maximal des lignes de chacune des tables est stocké et mis à jour. La clause Not peut alors se traiter comme suit (le but est de calculer l'arbre à radicaux $NotT$ avec T un arbre à radicaux.

On définit ici un n-arbre à radicaux complet un arbre à radicaux contenant toutes les valeurs de entiers compris entre 0 et $n-1$.

Pour calculer un "not", il suffit alors de retirer récursivement, à l'aide d'un $x-or$ les feilles de T d'un n arbre à radicaux (où n désigne l'indice maximal des lignes de la table d'expansion à laquelle appartient T).

Lorsqu'on retire un noeud d'un arbre à radicaux, on le retire et on retire récursivement son père s'il n'a plus de fils.

Par exemple, la figure 3 montre le calcul de $NotT$ lorsque la table d'expansion à laquelle T appartient à un indice maximum de lignes utilisé égal à 13.

Voici l'arbre initial (voir figure 3)
et l'arbre transformé (voir figure 4)

4.4 Comparaison entre colonnes

La comparaison entre deux colonnes est la plus complexe des requêtes atomiques. Cette requête se traite pratiquement comme un produit cartésien (voir la section suivante).

Soit `t` une table d'expansion et soient `t.c` et `t.d` deux de ses colonnes ? Une comparaison entre ces deux colonnes est une opération au cours de laquelle on souhaite discriminer les lignes de `t` pour lesquelles `t.c > t.d` par exemple. Nous insistons sur le fait que cette comparaison se fait à indice

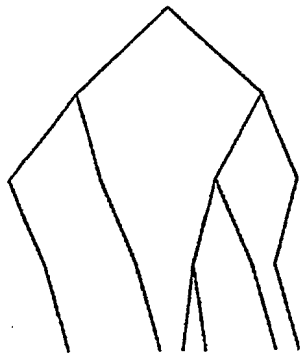

Figure 5: Le même après le NOT

par exemple. Nous insistons sur le fait que cette comparaison se fait à indice de ligne identique pour les deux colonnes, ce qui distingue cette comparaison du produit cartésien.

Comment pouvons nous alors effectuer cette requête ?

Nommons $\mathcal{T}_c$ et $\mathcal{T}_d$ les thesaurus des deux colonnes `t.c` et `t.d`. Nous recherchons les lignes telles qu'à ces lignes, `t.c > t.d`. Voici comment procéder. Pour tout mot w du thesaurus $\mathcal{T}_c$, on calcule l'arbre à radicaux r de l'intervalle $[m_d, w']$ où w' désigne le plus grand mot de $\mathcal{T}_d$ inférieur strictement à w. Alors en calculant un "et" sur r et l'arbre à radicaux de w, on obtient un arbre t_w. En effectuant lé réunion de tous les arbres à radicaux r_w, on obtient l'arbre à radicaux recherché.

Il est bien évident que les arbres t_w ne doivent pas être calculés indépendamment les uns des autres. Puisque les mots w sont parcourus dans l'ordre croissant, il suffit d'unir t_w à l'arbre correspondant à l'ajout des mots compris entre w et le mot suivant dans $\mathcal{T}_c$.

On peut aussi calculer `t.c - t.d` à l'aide des fichiers à plat et lire le résultat.

Les autres clauses semblables se résolvent de façon similaire (par exemple `t.c` $\geq$ `t.d`).

4.5 Le produit cartésien

Le produit cartésien est considéré habituellement comme une requête très combinatoire. En fait cette requête peut se résoudre assez simplement et efficacement. On peut en effet et à titre d'exemple calculer le produit cartésien de deux tables de 6 millions de lignes en moins de 7 secondes sur un ordinateur standart.

Une telle requête est une comparaison entre deux colonnes indépendamment des lignes auxquelles elles appartiennent. Par exemple, on peut vouloir

Par exemple, supposons que les colonnes `t.c` et `t.d` soient :

t.c
z
ab
z
c
da
e

et

t.d
b
a
as
sa
ca
ba
abra

Alors, le nombre de fois que `t.c > t.d` est $7+1+7+5+6+6 = 32$. La complexité de l'algorithme naïf est une constante fois le produit des tailles des deux tables. Il n'est donc en pratique pas possible de réaliser un produit cartésien sur des machines modernes en appliquant cet algorithme.

Les colonnes sont stockées avec leur thesaurus et les arbres à radicaux correspondants à chacune des entrées de ces thesaurus.

On peut alors, pour chacun des mots des thesaurus concernés par la requête, calculer le nombre d'apparitions de ce mot par lecture de l'arbre à radicaux correspondant.

t.c	
ab	1
c	1
da	1
e	1
z	2

t.d	
a	1
abra	1
as	1
b	1
ba	1
ca	1
sa	1

On peut aussi, pour la colonne `t.d`, calculer pour chacun des mots w du thesaurus le nombre d'apparitions d'un mot inférieur ou égal à w. Cela donne :

t.d, cumulated cardinalities	
a	1
abra	2
as	3
b	4
ba	5
ca	6
sa	7

Alors la lecture des thesaurus et les nombres d'apparitions des mots correspondants permet de calculer le résultat. En effet, pour chaque mot `w` de `t.c`, on recherche dans le thesaurus de `t.d` le plus grand mot `w'` inférieur ou égal à `w` et ajouter au résultat (initialisé à 0) le produit du nombre

d'apparitions de w par le nombre d'apparitions d'un mot inférieur ou égal à w.

Cela donne :

w	w'	w-card	w'-cumul. card.	product	partial result
ab	a	1	1	1	1
c	ba	1	5	5	6
d	ca	1	6	6	12
e	ca	1	6	6	18
z	sa	2	7	14	32

La complexité de cet algorithme est une constante fois la somme de la taille des thesaurus, qui est en général très inférieur au produit de la taille des table, même quand les colonnes sur lesquelles on réalise le produit cartésien ne contiennent aucun doublon.

4.6 Sous requêtes corrélées

Cette sous section et la suivante sont consacrées aux sous requêtes. En effet, il peut se trouver qu'une clause "where" contienne elle-même une autre clause "where", qui peut ou non être corrélée à la clause "where" principale.

Qu'est ce qu'une sous requête corrélée ? Un exemple d'une telle requête est donné par la requête 17 du TPC. Cette requête est :

```
select
     sum(l_extendedprice) / 7.0 as avg_yearly
from
     lineitem
     part
where
     p_partkey = l_partkey
     and p_brand = '[BRAND]'
     and p_container = '[CONTAINER]'
     and l_quantity < (
          select
               0.2 * avg(l_quantity)
          from
               lineitem
          where
               p_partkey = p_partkey
    );
```

Dans cette requête, on doit réaliser le calcul d'une sous requête tenant compte des conditions requises par la requête principale (puisque le p_partkey de la sous requête appartient à la clause where principale).

Donc ce genre de requête peut être réécrite de façon à changer cette sous requête en une sous requête non corrélée. Il suffit pour cela de dupliquer les conditions requises par la clause where principale dans la clause where de la sous requête corrélée. Dans notre exemple, cela donne :

```
select
     sum(l_extendedprice) / 7.0 as ag_yearly
from
     lineitem
     part
where
     p_partkey = l_partkey
     and p_brand = '[BRAND]'
     and p_container = '[CONTAINER]'
     and l_quantity < (
          select
               0.2 * avg(l_quantity)
          from
               lineitem
               partsupp
          where
               p_partkey = p_partkey
               and p_brand = '[BRAND]'
               and p_container = '[CONTAINER]'
     );
```

Finalement, une sous requête corrélée peut se réécrire en une sous requête non corrélée. C'est le sujet de la prochaine sous section.

4.7 Sous requêtes générales

Une requête SQL contenant des sous requêtes non corrélées peut se traiter en traitant d'abord les sous requêtes récursivement et en remplaçant dans la requête la sous requête par son résultat.

4.8 Effectuer des calculs aux lignes trouvées

Désormais, nous sommes en mesure de traiter une clause "where" qui nous renvoie un arbre à radicaux représentant les indices de lignes de la table d'expansion correspondant à cette clause.

Supposons à présent que l'objet de la requête ait été de réaliser certains calculs sur quelques colonnes aux lignes trouvées. Par exemple on peut

vouloir calculer la moyenne aux lignes trouvées des valeurs d'une certaine colonne.

Les valeurs de cette colonne sont stockées "à plat" dans leur ordre d 'apparition. Il est donc très simple de relire les valeurs de cette colonne uniquement pour les lignes trouvées précédemment et effectuer sur ces valeurs les calculs demandés.

5 Modifications de la base de données

Nous sommes ici en mesure de stocker toute une base de données et de l'interroger efficacement. Il reste encore à être capables de la modifier efficacement. La dialectique habituelle est que plus une base est gérée de façon à pouvoir être interrogée rapidement, plus les modifications sont coûteuses en temps et en ressources machines et vice versa.

Ce n'est pas le cas avec l'algorithme détaillé dans cette invention. Non seulement les requêtes sont résolues rapidement avec peu d'utilisation des ressources machines, mais cela reste vrai quand on modifie la base de données.

Transformer une base de données peut consister en ajouter ou retrancher des lignes, modifier des données, ajouter ou supprimer une clé primaire, ajouter ou supprimer une clé étrangère, ajouter ou supprimer une colonne de table, ajouter ou supprimer une table. Voyons successivement tous ces cas.

5.1 Modifier une table

Une modification de table est une opération des plus courantes quand on modifie une base de données. Il s'agit le plus souvent d'ajouter ou de retirer des lignes d'une table. Une autre transformation consiste en changer l'organisation de la base de données, mais ce genre de modifications est bien plus rare.

En général, lorsqu'on enlèvera des lignes d'une table, nous ne réorganiserons pas toute la table. Cela signifie que certains indices de ligne seront déclarés libres et que les données correspondantes seront effacées des tables d'expansion dans lesquelles elles sont présentes (voir plus loin). Une table comportera donc usuellement des "trous" : certaines lignes seront considérées comme vides de tout contenu, Les indices de ces lignes seront stockés et mis à jour.

5.1.1 Ajouter des lignes à une table

On souhaite donc ajouter des lignes à une table T. Nous insistons sur le fait que pour nous, la base n'est constituée que de tables d'expansion. On commence par compléter les lignes de T des colonnes des tables T^3 qui s'développent dans T.

Puis pour chaque table T' dans laquelle T s'développe, grâce au fichier des indices de lignes libres de T', nous attribuons à chaque ligne à ajouter un indice de ligne dans T'.

Enfin il ne reste qu'à calculer les arbres à radicaux de chacune des colonnes de la table T complétée, à réaliser un "or" avec les arbres à radicaux des mots correspondants dans T' et à stocker ces arbres en lieu et place des anciens.

`Oter des lignes à une table`

Il y a plusieurs problèmes à régler : toutes les tables de notre base de données sont des tables d'expansion et nous ne souhaitons pas réorganiser les tables de fond en comble après en avoir retiré certaines lignes.

Nous avons donc les indices de lignes devant être retirées d'une table T (ces indices peuvent avoir été trouvées à l'aide d'une clause where). Pour chaque colonne `t.c` de la table d'expansion T et pour chaque mot `w` on calcule l'arbre à radicaux de w pour les lignes à retirer qu'on x-or avec l'arbre de w avant modification. Le résultat est stocké en lieu et place de l'arbre initial.

On suppose qu'il n'est pas nécessaire de changer quoi que ce soit aux tables dans lesquelles T s'développe. En effet, si ce cas se présentait, cela signifierait que la clause `delete` était illégale.

5.1.2 Ajout d'une colonne

Ajouter une colonne c à une table d'expansion T consiste en plusieurs opérations. Il faut en effet traiter la table à laquelle la colonne appartient ainsi que toutes celles dans lesquelles cette table s'développe.

Le traitement de T consiste en la construction du thesaurus de c, et des arbres à radicaux correspondants ainsi que leur stockage.

Le traitement des tables T' dans lesquelles T s'développe consiste à lire les indices de lignes de T' auxquels doit être ajoutée la colonne c. Il suffit alors de calculer le thesaurus et les arbres à radicaux correspondants et de les stocker.

Prenons un exemple :

Tables reliées entre elles :

T0		
(li)	c1	fk1
(0)	a	2
(1)	b	1
(2)	c	0
(3)	b	1
(4)	e	2

T1			
(li)	pk1	c2	fk2
(0)	0	S	0
(1)	1	T	1
(2)	2	V	0

T2		
(li)	pk2	c3
(0)	0	X
(1)	1	Y

Les tables d'expansion sont donc :

développed T0							
(T0)			(T1)			(T2)	
(li)	c1	fk1	pk1	c2	fk2	pk2	c3
(0)	a	2	2	V	0	0	X
(1)	b	1	1	T	1	1	Y
(2)	c	0	0	S	0	0	X
(3)	b	1	1	T	1	1	Y
(4)	e	2	2	V	0	0	X

développed T1					
(T0)			(T1)		
(li)	pk1	c2	fk2	pk2	c3
(0)	0	S	0	0	X
(1)	1	T	1	1	Y
(2)	2	V	0	0	X

développed	T2	
(1i)	pk2	c3
(0)	0	X
(1)	1	Y

et supposons que nous souhaitions ajouter la colonne c2 à T2 dont les valeurs sont Y et Z.

La table T2 développée devient :

développed	T2		
(1i)	pk2	c3	c2
(0)	0	X	Y
(1)	1	Y	Z

Pour calculer la nouvelle table d'expansion T1, on mit les valeurs de la clé priamire pk2 et copie les valeurs correspondantes de T2 dans la nouvelle colonne de T1. Cela donne :

T1						
(T1)				(T2)		
(1i)	pk1	c2	fk2	pk2	c3	c2
(0)	0	S	0	0	X	Y
(1)	1	T	1	1	Y	Z
(2)	2	V	0	0	X	Y

De la même façon, on lit les valeurs de pk2 dans T0 pour calculer la nouvelle table d'expansion T0. Cela donne :

T0								
(T0)			(T1)			(T2)		
(li)	c1	fk1	pk1	c2	fk2	pk2	c3	c2
(0)	a	2	2	V	0	0	X	Y
(1)	b	1	1	T	1	1	Y	Z
(2)	c	0	0	S	0	0	X	Y
(3)	b	1	1	T	1	1	Y	Z
(4)	e	2	2	V	0	0	X	Y

5.1.3 Enlever une colonne à une table

Enlever une colonne à une table T est une opération très simple. Elle consiste simplement en l'effacement des fichiers correspondants pour cette colonne de la table T ainsi que pour toutes les tables dans lesquelles T s'développe.

5.1.4 Ajouter une clé primaire

Une clé primaire est stockée par les données des colonnes formant la clé ainsi que par une permutation permettant de retrouver l'ordre lexicographique dans lequel elles sont triées.

Ajouter une clé primaire revient donc à stocker d'une part les données des colonnes la formant avec la permutation correspondante.

5.1.5 Ajout d'une clé étrangère

Une clé étrangère, fk, appartenant à une table d'extension T, est attachée à une clé primaire, disons pk appartenant à une table d'extension T'. La table T' doit être développée dans T d'après le couple (fk, pk) même dans le cas où T' est déjà développée dans T par le biais d'un autre couple formé d'une clé étrangère et d'une clé primaire.

Pour toute ligne de T, d'indice i, la clé étrangère a une valeur v qui permet de retrouver l'indice de la ligne correspondante dans T', c'est-à-dire l'indice $p(i)$ de T' tel que pk[p(i)] = v.

On ajoute alors les colonnes de T' dans T. Pour réaliser cela, pour chaque colonne c de T', on lit la valeur T'.c[p[i]] pour chaque entier i. Puis on procède comme d'habitude en construisant le thesaurus et les arbres à radicaux de cette colonne qu'il ne reste qu'à stocker.

5.1.6 Retirer une clé primaire

Effacer une clé primaire se fait en effaçant les fichiers qui la décrivent. Si cette clé primaire est encore cible d'une clé étrangère, il faudrait naturellement lancer une exception car dans ce cas une telle demande est illégale.

5.1.7 Retirer une clé étrangère

Notons fk la clé étrangère à retirer et T la table à laquelle elle appartient. Cette clé étrangère cible une clé primaire, disons pk, appartenant à une table T'.

Retirer cette clé primaire brise ce lien entre T et T'. Cela signifie que T' perd cette extension dans T (il faut détruire les fichiers correspondants) ainsi que dans toutes les tables dans lesquelles T est développée.

5.2 Ajouter ou retirer une table de la base

Ajouter ou retirer une table de la base de données consiste à ajouter (ou retirer) toutes ses colonnes, toutes ses clés primaires et toutes ses clés étrangères. Tous ces procédés ont été vus plus haut.

6 Revendications

1 Un algorithme permettant une gestion complète d'une base de données en développant chacune des tables qui forment la base dans les tables auxquelles elle est reliée par un jeu de clés primaires clés étrangères. Cette étape consiste en :

```
Pour toute table T de la base de données relationnelle
  Pour toute clé étrangère f de T
    Ajouter les colonnes de T' où T' est une table ayant une
clé primaire cible de f
      Répéter ce procédé récursivement.
```

Une table dans laquelle toutes les tables qui pouvaient l'être ont été développées est nommée une table d'expansion.

2 Un algorithme dédié à la création des indexes.

Cette étape consiste en

```
Pour chaque colonne C de chaque table d'expansion T comme détaillé
dans la revendication 1
  Choisir un ensemble d'entiers positifs, les longueurs de troncatures.
  Créer et stocker le thesaurus T de la colonne C
    Pour chaque mot de ce thesaurus calculer l'ensemble des indices
de lignes auxquelles il apparaît.
    Créer et stocker l'arbre à radicaux dont les feuilles sont
les éléments de S
```

Pour toute longueur l de troncature

Créer et stocker le thesaurus T' de la colonne C dans laquelle tous les mots ont été tronqués à l caractères (si C est de type chaîne de caractères ou dates ...) ou de l caractères (si la colonne C contient des données de type numérique).

Pour tout mot de T' calculer l'ensemble S' des indices de lignes auxquels il apparaît.

Créer et stocker l'arbre à radicaux dont les feuilles sont les éléments de S'.

3 Une étape consistant en stocker à plat les données de chaque colonne de chaque table d'expansion.

Cette étape consiste en général à stocker la suite des indices des mots dans le thesaurus dans l'ordre dans lequel ils apparaissent dans la colonne.

4 Un algorithme dédié aux requêtes SQL. Cette étape consiste en

Résoudre la clause "where". Cette étape renvoie un arbre à radicaux.

Calculer la table d'expansion dans laquelle doit être résolue la requête SQL. Cette étape se fait en calculant l'ensemble S des tables dans lesquelles sont développées les tables impliquées dans la requête. Parmi les table de S, une seule n'est pas développée dans toutes les autres. C'est la table d'expansion pertinente pour la requête à résoudre. À partir d'ici, la référence à la table d'expansion signifiera cette table.

Effacer les clauses de jointure de la requête.

Résoudre récursivement les sous requêtes. Dans le cas d'une sous-requête corrélée, les conditions de la clause ''where'' dont elle dépend seront dupliquées dans la sous requête qui sera alors traitée comme une sous requête non corrélée. Lorsque la sous requête n'est pas corrélée, elle est traitée comme une requête indépendante de la requête principale et son résultat la remplace au sein de la requête principale.

Résoudre la clause ''where'' elle même.

La résolution d'une requête ne contenant pas de sous requête est expliquée plus bas.

5 Un algorithme pour résoudre les requêtes SQL ne contenant pas de sous requête.

Reconnaître et résoudre les parties atomiques de la requête. Une partie atomique de requête est une partie de la where clause ne contenant pas de connecteur logique comme ''et'', ''ou'' ou ''not''.

Mixer les résultats des requêtes atomiques au moyen des clauses ''and'', ''or'', ''not'' présentes dans la clause ''where''.

6 Un algorithme pour résoudre les requêtes SQL

- Une clause d'égalité entre une colonne et une constante est traitée en lisant l'arbre à radicaux de la constante.

- Une clause ''between'' est traitée en séparant l'intervalle demandé en intervalles dont les bornes sont de longueur les longueurs de troncatures de la colonne de façon à minimiser le nombre d'arbres à radicaux à lire.
- Une clause ''in'' est traitée comme des clauses d'égalité séparées par des ''or''.
- une clause de comparaison entre une colonne et une constante ($\geq$, $>$, $<$, $\leq$) est traitée comme un between en utilisant comme borne la valeur minimale ou maximale du thesaurus.
Une clause ''like'' est traitée comme une clause ''between''.
- Une comparaison entre deux colonnes est traitée en utilisant les thesaurus des colonnes et les arbres à radicaux des colonnes ou les fichiers à plat des colonnes comparées.
Pour le premier cas, Nommons $\mathcal{T}_c$ et $\mathcal{T}_d$ les thesaurus des deux colonnes t.c et t.d. Nous recherchons par exemple les lignes telles qu'à ces lignes, t.c > t.d. Pour tout mot w du thesaurus $\mathcal{T}_c$, on calcule l'arbre à radicaux r de l'intervalle $[m_d, w']$ où w' désigne le plus grand mot de $\mathcal{T}_d$ inférieur strictement à w. Alors en calculant un ''et'' sur r et l'arbre à radicaux de w, on obtient un arbre t_w. En effectuant la réunion de tous les arbres à radicaux r_w, on obtient l'arbre à radicaux recherché.
On peut aussi calculer t.c - t.d à l'aide des fichiers à plat et lire le résultat.
Les autres clauses semblables se résolvent de façon similaire (par exemple t.c $\geq$ t.d).

7 Un algorithme pour résoudre la clause "where" de la requête.

Cette étape consiste à traiter récursivement les jonctions logiques entre les sous requêtes de la clause where.

8 Une étape consistant à effectuer des calculs sur les données

Cette étape consiste à lire les fichier à plat des colonnes impliquées dans la formule de calcul et de réaliser les opérations demandées.

9 Une étape consistant à effectuer des calculs sur les données

Dans le cas d'un produit cartésien par exemple on peut utiliser les thesaurus et les cardinaux des arbres à radicaux, cumulés ou non. Si on veut dénombrer par exemple le nombre de fois que t.c > t'.d, pour chaque mot w de t.c, on recherche dans le thesaurus de t.d le plus grand mot w' inférieur ou égal à w et ajouter au résultat (initialisé à 0) le produit du nombre d'apparitions de w par le nombre d'apparitions d'un mot inférieur ou égal à w.

10 Un algorithme consistant à gérer la base de données

Les modifications de la base de données peuvent consister à ajouter ou retirer des lignes à une table, mettre à jour du contenu de table, ajouter ou retirer une colonne de table, ajouter ou retirer une clé primaire ou une clé étrangère, ajouter ou retirer une table.

10.1 Ajouter des lignes à une table T consiste à
Assigner un indice de ligne libre à chaque ligne à ajouter
Pour chaque colonne de la table
Calculer le thesaurus des lignes à ajouter et l'arbre à radicaux de chacun des mots à apparaissant dedans.
Ajouter les nouveaux mots du thesaurus de la colonne et calculer et stocker tw ou nw où tw est l'arbre à radicaux du mot w (vide si w n'était pas présent dans cette colonne et nw l'arbre à radicaux des indices de lignes auxquelles w est ajouté.

10.2 Retirer des lignes à une table consiste à :
Pour chaque colonne de la table
calculer le thesaurus T des lignes à retirer et l'arbre à radicaux de chacun des mots de ce thesaurus pour les lignes en question
Pour chaque mot w de T faire un X-or de l'arbre à radicaux de w de la base avec celui qui vient d'être calculé et remplacer l'arbre à radicaux de w décrivant sa présence dans la colonne par le résultat de ce calcul.

10.3 Mettre à jour du contenu consiste à retirer les contenus à remplacer comme s'il s'agissait des lignes de la table à laquelle il appartient et ajouter le contenu comme s'il s'agissait de lignes à ajouter.

10.4 Ajouter une colonne C à une table T consiste à :
Calculer le thesaurus et les arbres à radicaux de la colonne
Les stocker
Pour chaque table T' dans laquelle T se développe :
Construire la colonne développée dans T' en respectant les chemins entre T' et T via les jeux de clés primaires clés étrangères.
Ajouter la colonne construite à T' comme cela a été fait pour T

10.5 Retirer une colonne C à une table T consiste à effacer son thesaurus et ses arbres à radicaux dans T ainsi que dans toutes les tables T' dans lesquelles T est développée.

10.6 Ajouter une clé primaire à une table T consiste à stocker à plat ses valeurs et une permutation permettant d'en lire les valeurs dans l'ordre lexicographique ou ordinal.

10.7 Effacer une clé primaire consiste à effacer son stockage à plat ainsi que la permutation correspondante.

10.8 Ajouter une clé étrangère f (ciblant une clé primaire d'une table T') à une table T consiste à ajouter dans T toutes les colonnes de la table développée T' en respectant les valeurs de f et répéter cette opération pour toutes les tables dans lesquelles T est développée.

10.9 Effacer une clé étrangère f (ciblant une clé primaire d'une table T') à une table T consiste à effacer dans T toutes les colonnes de la table développée T' en respectant les valeurs de f et répéter cette opération pour toutes les tables dans lesquelles T est développée.

10.10 Ajouter ne table consiste à ajouter ses colonnes, ses clés primaires et ses clés étrangères.

10.11 Effacer ne table consiste à effacer ses colonnes, ses clés primaires et ses clés étrangères.

1er dépôt

date de dépot
22.10.04